Stos – struktura danych:

- typu *Last In First Out* (LIFO) - ostatni wprowadzony element opuści ją jako pierwszy,
- umieszczona w pamięci operacyjnej komputera,
- adresowana **wierzchołkiem** (wskaźnikiem) **stosu (stack pointer)**:
  zapewniony jest bezpośredni, łatwy dostęp do ostatniego odłożonego elementu
  (ale można odczytać i modyfikować dowolny element umieszczony na stosie).

W ogólnym przypadku obsługiwana z reguły instrukcjami:

PUSH: umieszcza element na stosie,

POP (ew. PULL): ściąga ostatni element ze stosu.

Stos – architektura x86-64

- **stos jest opadający (descending stack)**
  tzn. „rośnie w dół” – w kierunku malejących adresów,

- **wierzchołek stosku** (64-bitowy %rsp) **wskazuje na adres ostatniego odłożonego elementu**,

- w trybie 64-bitowym, na stos można odkładać:

  16- i 64-bitowe rejestry i zmienne z pamięci RAM,
  (8-), 16- i 32-bitowe (rozszerzane ze znakiem do 64 bitów) stałe.

  i ściągać 16- i 64-bitowe słowa do rejestrów lub lokalizacji w pamięci RAM.

Np.: 64-bitowe instrukcje **PUSH** i **POP** wykonują operacje (na rejestrze):

- **PUSH** %reg64

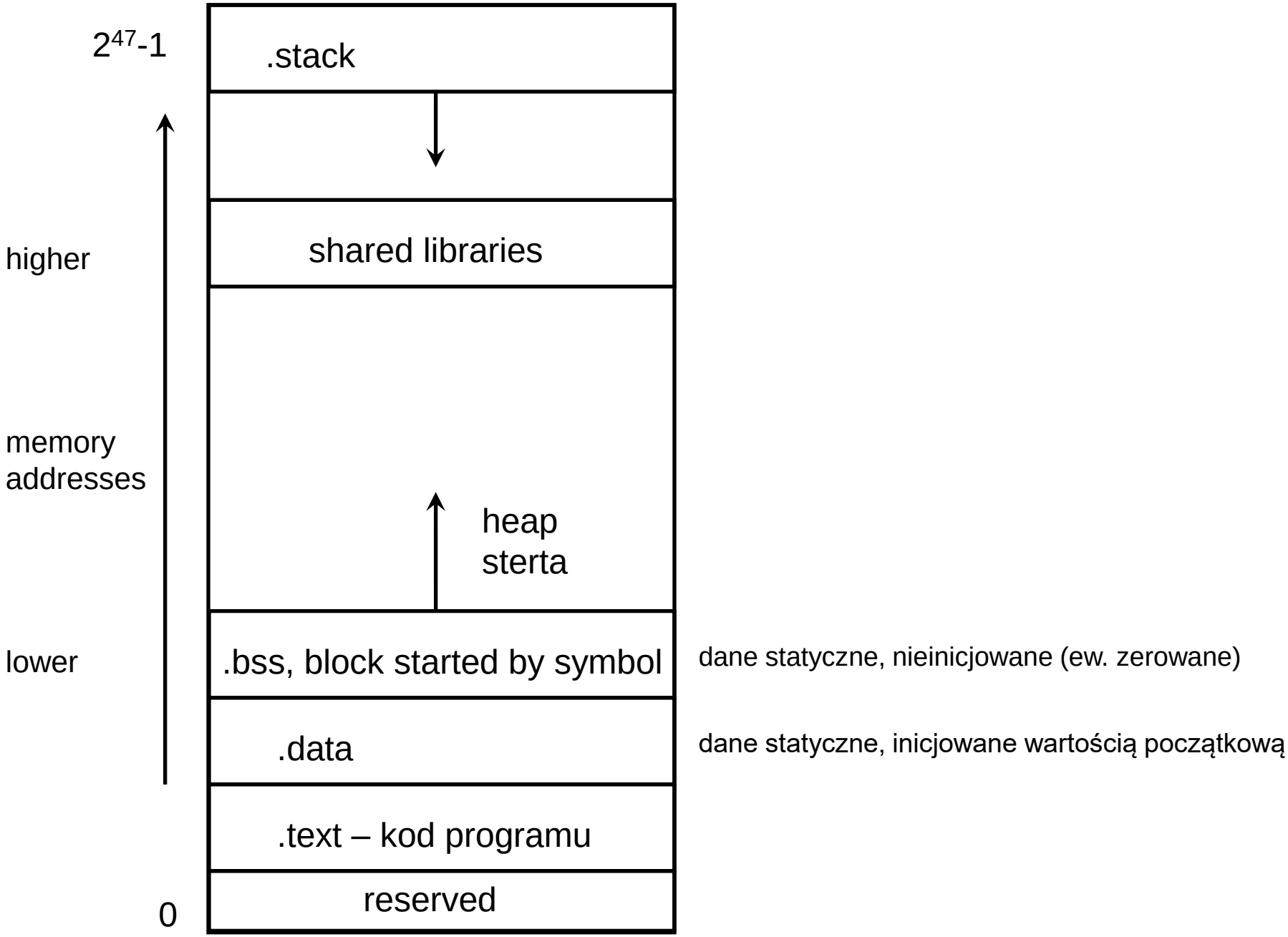
```
1. SUB $8 , %rsp
2. MOV %reg64,(%rsp)
```

- **POP** %reg64

```
1. MOV (%rsp),%reg64
2. ADD $8 , %rsp
```

w podobny sposób odkładają i ściągają adres powrotny instrukcje CALL i RET

Uproszczona mapa pamięci programu – Linux, 64-bitowy



Przekazywanie parametrów przez stos, alokacja zmiennych lokalnych (przykład 32-bitowy!)

```
int main(void)
{
   …
   subroutine(arg1, arg2, arg3);
   …
   return(0);
}
```

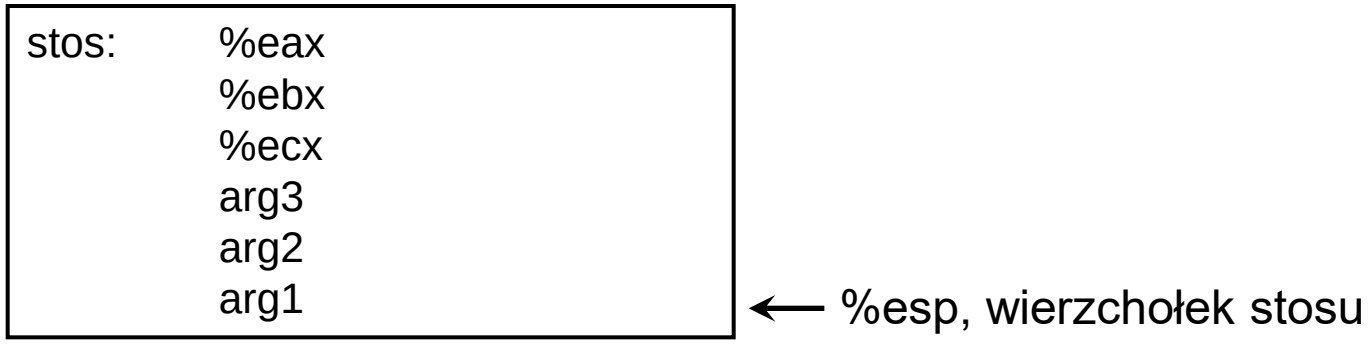
```
void subrouitne(int arg1, int arg2, int arg3)
{ int local_1, local_2;
…
}
```

Założenia:

- Zarówno funkcja wywołująca (main) jak i wywoływany podprogram (subroutine) wykorzystują te same rejestry: %eax, %ebx i %ecx do własnych celów.
- Rejestry %eax, %ebx, i %ecx nie są zachowywane przez podprogram (musi to zrobić main).
- Podprogram przechowuje **dwie zmienne lokalne typu int na stosie**.

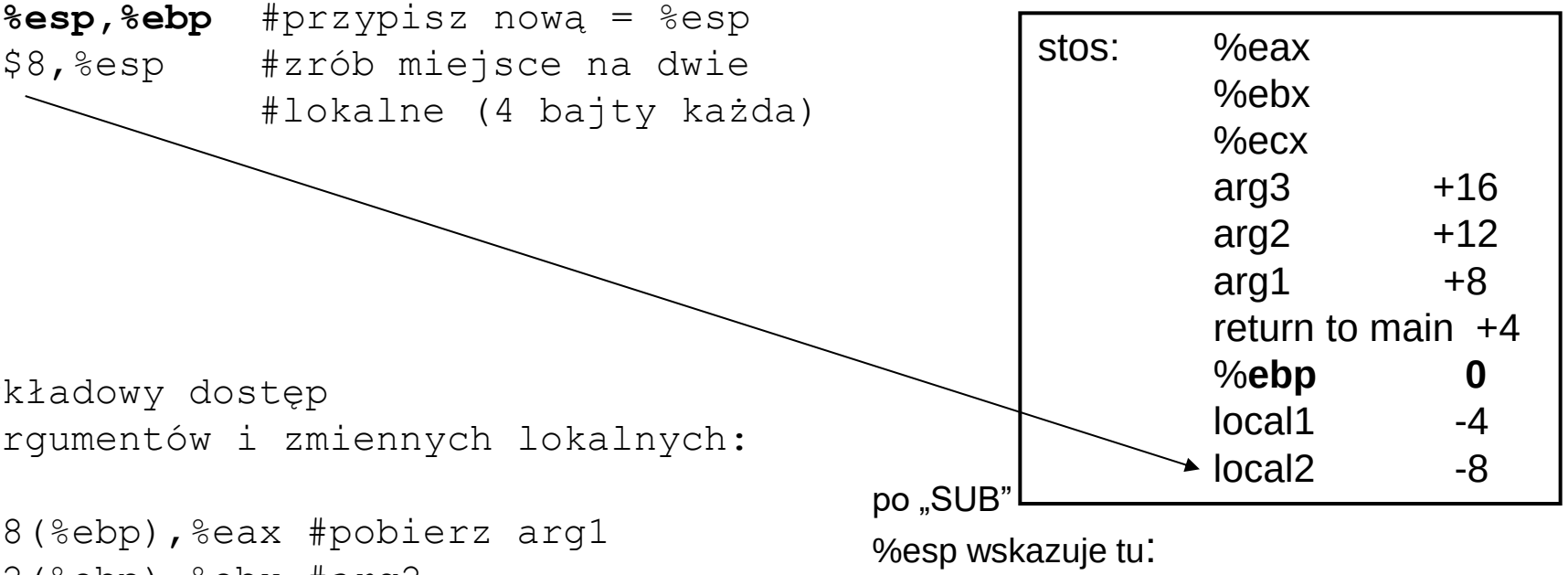
(1) przed wywołaniem funkcji

```
main:
…
PUSH %eax #zachowaj wartośći w rej.
PUSH %ebx #%eax, %ebx and %ecx
PUSH %ecx #używane przez main

PUSH arg3 #umieść na stosie
PUSH arg2 #argumenty
PUSH arg1 #zaczynając od ostatniego

CALL subroutine
```



(2) po wejściu w podprogram

```
subroutine:
PUSH %ebp          #zachowaj starą ramkę stosu (z main)
MOV  %esp,%ebp     #przypisz nową = %esp
SUB  $8,%esp       #zrób miejsce na dwie
…                  #lokalne (4 bajty każda)
```




```
#przykładowy dostęp
#do argumentów i zmiennych lokalnych:

MOV  8(%ebp),%eax #pobierz arg1
MOV 12(%ebp),%ebx #arg2
MOV 16(%ebp),%ecx #i arg3 ze stosu
…
MOV -4(%ebp),%eax #pobierz local1
MOV -8(%ebp),%ebx #pobierz local2

…
MOV %ebp,%esp  #zwolnij pamięć po zmiennych
               #lokalnych
POP %ebp       #przywróć mainowi jego ramkę stosu
RET
```

(3) po wejściu w podprogram

```
subroutine:
PUSH %ebp          #zachowaj starą ramkę stosu (z main)
MOV  %esp,%ebp     #przypisz nową = %esp
SUB  $8,%esp       #zrób miejsce na dwie
…                  #lokalne (4 bajty każda)
```

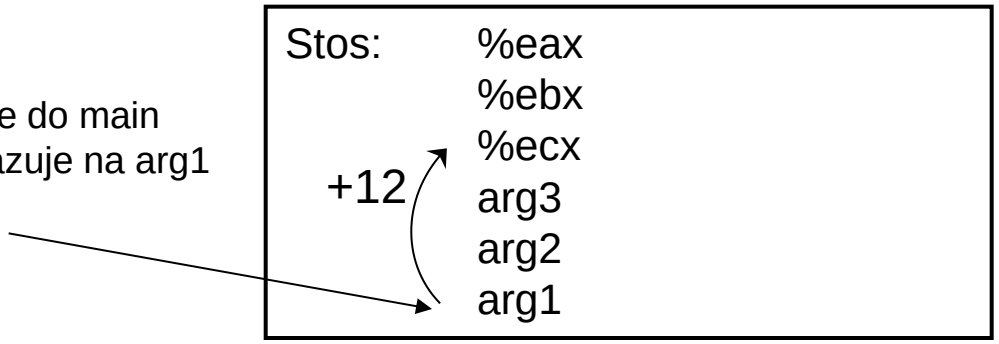
Stos:		
	%eax	
	%ebx	
	%ecx	
	arg3	+16
	arg2	+12
	arg1	+8
	return to main	+4
	%ebp	**0**
	local1	-4
	local2	-8

```
#przykładowy
#dostęp do argumentów i zmiennych lokalnych:

MOV  8(%ebp),%eax #pobierz arg1
MOV 12(%ebp),%ebx #arg2
MOV 16(%ebp),%ecx #i arg3 ze stosu
…
MOV -4(%ebp),%eax #pobierz local1
MOV -8(%ebp),%ebx #pobierz local2

…
MOV %ebp,%esp   #zwolnij pamięć po zmiennych
                #lokalnych
POP %ebp        #przywróć mainowi jego ramkę stosu
RET             #teraz %esp wskazuje na adres powrotny
```

czyli: przesuń %esp
(gdziekolwiek byłby)
na miejsce gdzie jest zachowana
poprzednia wartość %ebp

(4) powrót do main:

```
main:

CALL subroutine

ADD $12,%esp    #zwolnij pamięć na
                #stosie po
                #trzech argumentach

POP %ecx        #przywróć oryginalne
POP %ebx        #wartości rejestrom
POP %eax        #eax, ebx i ecx

...
```



• Instukcje typu: **POP**, **RET**, **IRET** nie usuwają fizycznie danych ze stosu, przesuwają jedynie wskaźnik. Dane pozostają w pamięci, aż zostaną nadpisane podczas odkładania kolejnych elementów.

• Instrukcje operujące na stosie wykonują zwykły dostęp do pamięci – tym samym spowalniają wykonywanie programów.

Ramka stosu

- **wskaźnik stosu** z reguły **zmienia swoje położenie** w trakcie wykonywania programu, co za tym idzie przesunięcie (liczba bajtów) między jego aktualnym położeniem i danymi na stosie również się zmienia,

- utworzona dla każdego* podprogramu **ramka stosu** (w rejestrze %ebp/rbp) zapewnia **stały punkt odniesienia**, pozwalający na łatwiejszy dostęp do umieszczonych na stosie elementów innych niż ostatnio odłożony, np. zmiennych lokalnych i argumentów. Przesunięcie między nimi a ramką stosu **jest stałe**.

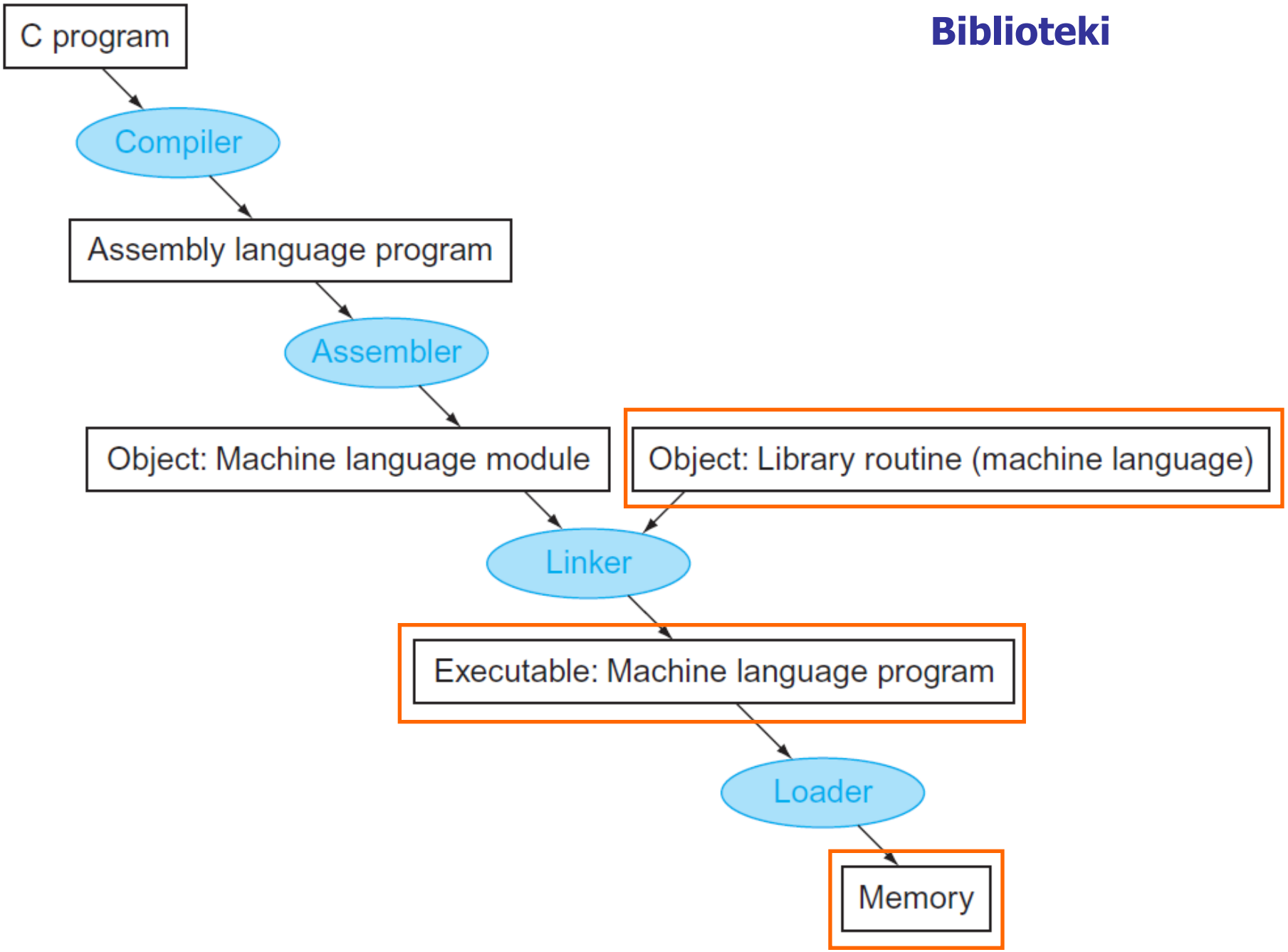
* w przypadkach prostych programów oraz przechowywania zmiennych lokalnych w rejestrach ramki stosu nie tworzy się (podobnie postępuje również kompilator).

Należy pamiętać, że zgodnie z ABI 64 bit przed wywołaniem funkcji bibliotecznych C wierzchołek stosu musi być wyrównany do granicy 16 bajtów (8 adres powrotu + 8 ramka stosu).

Jeżeli ramki stosu nie tworzymy, wskaźnik stosu najlepiej przesunąć szybką operacją arytmetyczną (+/- 8) niż odkładać (ściągać) jakiś argument do (z) pamięci.

Dzięki temu standard jest zachowany oraz unikamy dwóch dostępów do pamięci.

Biblioteki


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Bilibioteki dynamicznie ładowane i współdzielone (Dynamically Linked Libraries – DLL, Shared Objects - SO)

Lazy/Late Binding (Loading, Linking…) itp.

- podczas uruchamiania ładowany do pamięci jest tylko program główny oraz tablice dołączone do w plików wykonywalnych ELF (Linux):

  - *Procedure Linkage Table* (PLT),
  - *Global Offset Table* (GOT), ew. procedura linkera dynamicznego,

- kod danej funkcji bibliotecznej ładowany jest do pamięci **dopiero podczas pierwszego wywołania** tej funkcji w programie.

Zasada działania:

wszystkie wywołania funkcji bibliotecznych linkowanych dynamicznie odbywają się pośrednio: poprzez tablicę PLT oraz tablicę GOT – zawierającą adresy funkcji.

Pierwsze wywołanie funkcji:

1) Instrukcja CALL nie wywołuje bezpośrednio funkcji, ale prostą procedurę (*stub*) w tablicy PLT. Rozkaz CALL zachowuje na stosie adres powrotu do programu wywołującego daną funkcję.

2) Wykonywany jest skok pod adres odczytany z odpowiedniego miejsca tablicy GOT.

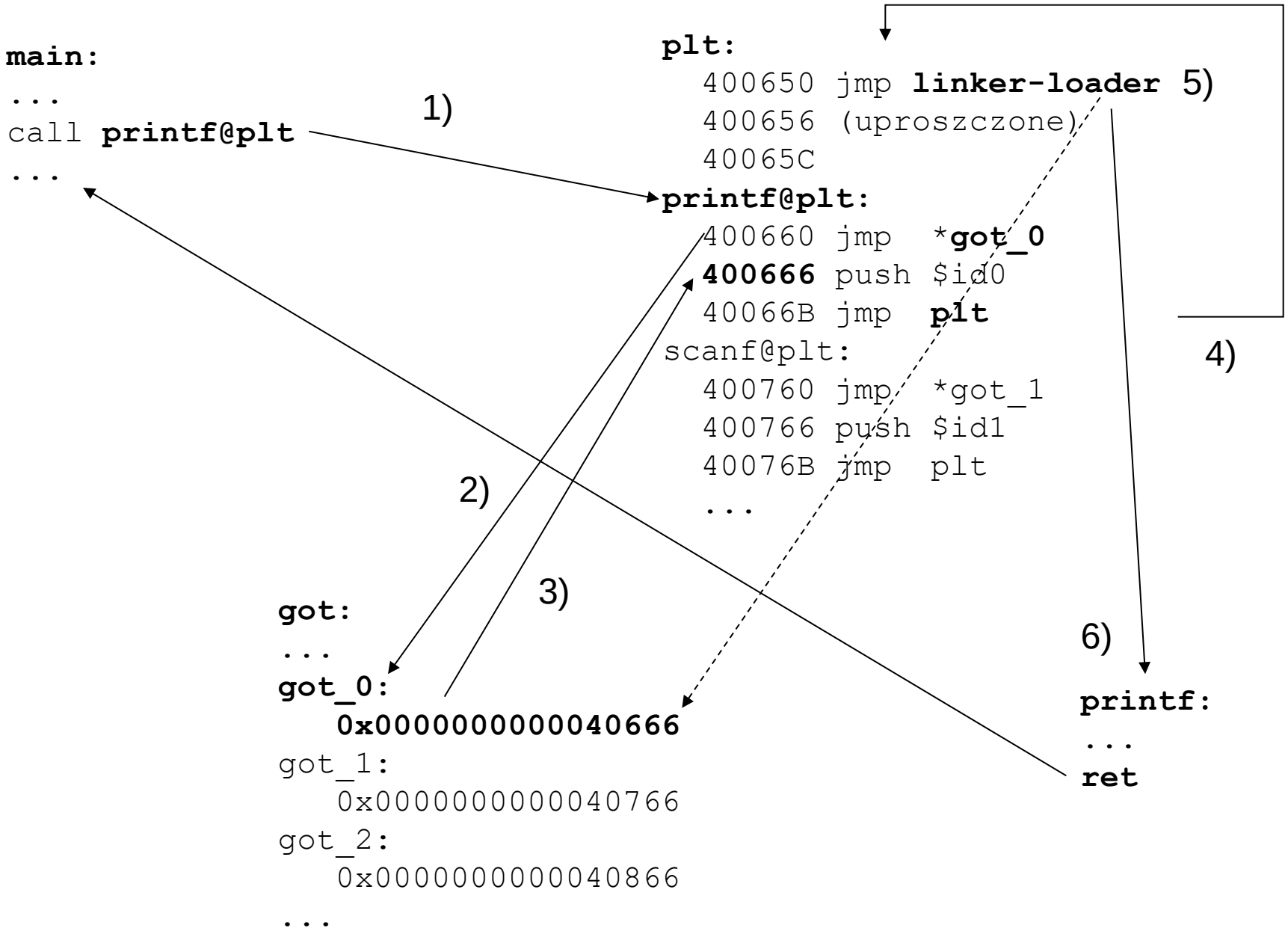
3) Przy pierwszym wywołaniu danej funkcji adresem tym jest adres kolejnej instrukcji w PLT.

4) Na stosie umieszczany zostaje numer-identyfikator żądanej funkcji bibliotecznej, następnie wywoływany jest linker dynamiczny.

5) Żądana funkcja jest ładowana z pliku do pamięci, a odpowiadający jej adres w GOT zamieniany na właściwy: prowadzący do miejsca, gdzie funkcję załadowano.

6) Funkcja zostaje wykonana, kończy ją rozkaz RET i następuje powrót i kontynuacja programu wywołującego.

Stan początkowy – przed pierwszym wywołaniem funkcji



Zasada działania:

drugie i kolejne wywołania tej samej funkcji:

1) Instrukcja CALL ponownie nie wywołuje bezpośrednio funkcji, ale wykonuje skok do PLT. Rozkaz zachowuje na stosie adres powrotu do programu wywołującego funkcję.

2) W PLT wykonywany jest skok pod adres odczytany z odpowiedniego miejsca tablicy GOT.

3) Przy kolejnym wywołaniu funkcji adresem skoku jest już miejsce poprzednio załadowanej funkcji.

4) Funkcja zostaje wykonana, ponieważ już rezyduje w pamięci, drugie i kolejne wywołania przebiegają szybciej od pierwszego.

Kolejne wywołanie tej samej funkcji bibliotecznej